Distributed Implementation of SIGNAL: Scheduling & Graph Clustering

Olivier Maffeis, Paul Le Guernic

Distributed Implementation of SIGNAL: Scheduling & Graph Clustering*

Olivier MAFFEÏS[1] and Paul LE GUERNIC[2]

[1] GMD I5 - SKS, Schloss Birlinghoven, 53754 Sankt Augustin, Germany
[2] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract. This paper introduces the scheduling strategy and some key tools which have been designed for the distributed implementation of SIGNAL, a real-time synchronous dataflow language. First, we motivate a scheduling strategy with respect to the reactivity and time-predictability requirements bound to real-time computing. Then, several key tools to implement this scheduling strategy are described. These tools are acting on the concept of Synchronous-Flow Dependence Graph (SFD Graph) which defines a generalization of Directed Acyclic Graph and constitutes the abstract representation of SIGNAL programs. The tools presented in this paper are: (a) the abstraction of SFD graphs which enables grain-size tuning according to the target architecture, (b) the notion of scheduling over SFD graphs and (c) qualitative clustering tools based on the notion of *Compositional Deadlock Consistency*.

1 Introduction

Although distributed architectures are becoming increasingly popular, implementing large-scale applications onto them still remains a very difficult challenge. The problem of implementing a program onto a distributed architecture is often stated as: partitioning and scheduling the nodes of a Directed Acyclic Graph (DAG) $< N, \Gamma >$ onto a set of potentially heterogeneous processors $\{P_i \,|\, i = 1, \ldots, p\}$. In the abstract representation $< N, \Gamma >$ of the application, N is a set of nodes (tasks) which stand for indivisible[3] program computations, and Γ is a set of arcs which represent the precedence constraints (including the data paths). If the scheduling goal is the minimization of the program completion time, the associated scheduling problem is NP-complete even if the number of processors is unbounded [18]. Since the optimal solution of this scheduling problem can only be computed by exponential complexity algorithms (unless P=NP), the purpose is to define fast heuristic techniques which efficiently compute optimal or near-optimal solutions for restricted scheduling problems, i.e.

* This work has been initiated at IRISA (Institut de Recherche en Informatique et Systèmes Aléatoires). It has been completed at RAL (Rutherford Appleton Laboratory, England) and GMD where it has been supported by an ERCIM (European Research Consortium for Informatics and Mathematics) fellowship.

[3] in the sense that no attempt is made to use intranode parallelism

fast heuristic techniques acting on a previously stated scheduling strategy.

In this paper, we present the scheduling strategy and some key tools we designed for the distributed implementation of SIGNAL [13], a real-time synchronous data-flow language. The scheduling strategy we chose is presented in section 2; it takes into account responsiveness, robustness, efficiency and time-predictability requirements that real-time implementations must satisfy.

The tools implementing this scheduling strategy are acting on a generalization of the notion of Directed Acyclic Graphs called Synchronous-Flow Dependence Graphs (SFD Graphs). These graphs, which have been initially designed to represent abstractly SIGNAL programs, constitute now the graph format shared by the languages ESTEREL [5], ARGOS [17], LUSTRE [8] and SIGNAL; SFD graphs are presented in section 3.

The following three sections define three key tools over SFD graphs to implement our scheduling strategy: (a) the abstraction of SFD graphs enabling grain-size variations in section 4, (b) the notion of compile-time scheduling over SFD graphs in section 5 and, (c) clustering tools based on a new qualitative criterion, namely the *compositional deadlock consistency*, in section 6.

2 Scheduling Strategy

The possible scheduling strategies go from fully dynamic up to fully static. In the fully dynamic scheduling strategy, the assignment of the tasks to the processors and their firing time are determined at run-time. In contrast, the fully static approach realizes these two actions at compile time. Static or dynamic task assignment is the first criterion which motivates the choice of a scheduling strategy. A dynamic assignment scheduling strategy induces efficient parallel program executions if the target architecture has relatively low communication costs compared with the processor performance; a fully dynamic scheduling strategy performs well for shared-memory architectures with few processors but not for large-scale distributed memory architectures. Due to the relatively weak scalability of the dynamic assignment scheduling strategies, we chose a scheduling strategy with a static assignment of the tasks.

2.1 Related Work

Many works on static assignment scheduling strategies have been achieved but they often consider only particular target architectures (the processors are homogeneous, the network topology is a ring, etc.) or particular application graphs (chains, trees, etc.) [3] for which polynomial optimal algorithms can be found.

One relevant approach for a general purpose multiprocessor scheduling strategy has been proposed by Kim & Browne [9]. Their scheduling strategy decomposes the application graph into a set of linear clusters. A linear cluster is a set of nodes in which, for every couple of nodes, one precedes the other. The clustering

algorithm merges iteratively the nodes on the critical path. After some clustering refinements, each cluster is mapped to one processor of the target architecture.

In [18], Sarkar has proposed a similar scheduling strategy: a clustering phase (called internalization prepass) followed by a mapping phase. His clustering algorithm considers the arcs of the application graph in a descending order with respect to their communication weight. It iteratively merges the extremity nodes of the first arc in the list if this clustering does not increase the parallel completion time; this clustering algorithm performs non linear clustering. When the arc list has been exhausted, the mapping phase is achieved using a list scheduling algorithm.

Another relevant approach has been defined by Gerasoulis & Yang in [7]. This approach, implemented in the PYRROS environment [20], uses a clustering algorithm which merges the extremity nodes of the highly weighted arc on the Dominant Sequence path. The Dominant Sequence path is the longest (i.e. critical) path in the estimated static scheduling[4]. Like in Sarkar's approach, this algorithm performs non-linear clustering.

2.2 The SIGNAL Approach

To implement SIGNAL programs on a distributed architecture with p processors, we advocate a slightly different scheduling strategy:

(a). Gather the nodes of the application graph into u clusters ($u \geq p$).
(b). Merge the u clusters into p connected virtual processors.
(c). Map the p virtual processors onto the p physical processors.
(d). Partition each virtual processor i in v_i clusters.
(e). Compute a static schedule for each cluster; the resulting sequences of code will be dynamically scheduled.

In contrast with the other scheduling strategies which are fully static, our strategy envisages mixed static/dynamic scheduling at the processor level (step (e)). This modification has been motivated by the kind of applications we have to cope with: reactive instead of transformational systems [16]. In transformational systems, the inputs are defined before the execution of the system. Therefore, an implementation of this system may schedule the reading of its inputs at compile time; a fully static scheduling strategy may induce efficient implementations for transformational systems. As we consider real-time systems, a timely kind of reactive systems, the inputs are supplied at run-time and we have to cope with robustness, responsiveness and time-predictability. Therefore, a scheduling strategy with some dynamic scheduling is more suitable for the implementation of real-time systems since it enables some flexibility in task firing. The dynamic scheduling is intended to provide the implementation with responsiveness and robustness to the variations in time of the input occurrences. But, this dynamic scheduling must be strictly confined to satisfy the time-predictable requirement.

[4] Note that Critical Path and Dominant Sequence path are equivalent notions in a linear clustering algorithm.

Note that, in our scheduling strategy, we have extended the notion of cluster. Usually, a cluster is defined as a set of tasks which are executed on the same processor. In the expression of our scheduling strategy, we only consider a cluster as a set of tasks which are treated as a whole in the next non-clustering steps. In the distribution steps (steps (a), (b) and (c)), this induces that all the tasks of a cluster will be implemented on the same processor. In the implementation steps (steps (d) and (e)), it signifies that dynamic scheduling will be only considered between the clusters, a sequence of code being associated with each cluster v_i.

In the sequel of this paper, we do not pretend to provide a complete set of tools to implement our scheduling strategy, but only to present some *key qualitative tools*:

- *abstraction of SFD graphs* in section 4. With this abstraction, SFD graphs may constitute the only modeling along the inference of a parallel implementation;
- *compile-time scheduling of SFD graphs* in section 5. With this notion, a first step towards the inference of parallel implementations over SFD graphs is achieved.
- *clustering tools based on new qualitative scheduling criterion:* **compositional deadlock consistency** in section 6. These clustering tools may be used in the implementation of steps (a) and (d) of our scheduling strategy.

Before presenting all these tools, let us present shortly the notion of Synchronous-Flow Dependence Graph (SFD Graph).

3 Synchronous-Flow Dependence Graphs

Let us illustrate the notion of Synchronous-Flow Dependence Graph (SFD Graph) over a simple SIGNAL example, a counter with reset:

```
(| ZV:= V $1
 | V:= ( 1 when RST )default( ZV+1 )
 |)
```

The `$` operator is used to recall past values: the process `ZV:= V $1` means that `ZV` carries the previous value of `V`. The `when` operator filters data according to a boolean condition and the `default` is a merge with priority:

```
V:= ( 1 when RST )default( ZV+1 )
```

specifies that `V` is reset to 1 when `RST` holds *true*; otherwise it increments its previous value. Sequences of values that `RST`, `ZV` and `V` may take are:

RST	:	$\cdots$				t						$\cdots$
ZV	:	$\cdots$	5	6	7	8	9	1	2	3	4	5 $\cdots$
V	:	$\cdots$	6	7	8	9	1	2	3	4	5	6 $\cdots$

This behavior is abstractly represented in two connected structures: an equation system which translates the relations among the occurrences of data, and a dependence graph which represents the flows of data.

An Equational Control Modeling

The equation system is expressed over a set C of characteristic functions called *clocks*: for any signal `V`, its clock $\widehat{v}$ is equal to 1 if `V` is carrying a data at the considered instant, it is equal to 0 if no data is present on `V`. The logical relations among the occurrences of data, which are implicit in SIGNAL processes, are translated into *equations over clocks*. For instance, the counter example is translated as:

```
ZV:= V $1                                   ẑv = v̂        (i)
V:= ( 1 when RST )default( ZV+1 )           v̂ = r̂st ∨ ẑv   (ii)
```

As expressed in equation (i), each time `V` is holding a value, `ZV` is carrying a value (in fact the previous value of `V`). Equation (ii) expresses that `V` carries a value whenever a reset occurs or `ZV` holds a value. Formally, the equation system Σ which encodes the occurrence relations evolves in a boolean algebra $\mathcal{B} =< C, \vee, \wedge, \widehat{0}, \widehat{1} >$ where:

- C is a set of clocks; $\mathcal{B}$ is called a clock algebra;
- $\widehat{0}$ denotes the least element of $\mathcal{B}$ which stands for the never present clock; it is used to denote something that never happens;
- $\widehat{1}$ is the greatest element of $\mathcal{B}$, the always present clock.

As boolean algebras are lattices, an alternative representation of the clock algebra $\mathcal{B}$ is achieved through a partial order:

$$< C, \leq > \qquad \text{with} \qquad \widehat{x} \leq \widehat{y} \iff \widehat{x} \vee \widehat{y} = \widehat{y} \quad (\Leftrightarrow \ \widehat{x} \wedge \widehat{y} = \widehat{x})$$

Over the counter encoding, we can deduce that $\widehat{v} = \widehat{rst} \vee \widehat{v}$ or equivalently $\widehat{rst} \leq \widehat{v}$. This result intuitively means that the activity of the counter includes the reset operations.

A Clock-Labeled Dependence Graph

The equation system describes algebraically the reachable control states of the process. Over the counter example, the relation $\widehat{rst} \leq \widehat{v}$ induces that the control state where $\widehat{v} = 0, \widehat{rst} = 1$ is unreachable. According to the reachable control states, different flows of data may occur; the different flows of data which may occur in the counter are abstractly represented by the dependence graphs in Fig. 1-a, Fig. 1-b and Fig. 1-c, one for each reachable control state.

In Fig. 1-a where `RST` occurs ($\widehat{rst} = 1$), `ONE` (a constant signal) is assigned to `V`, the value of `ZV` is not used. Otherwise, `V` is defined[5] by the value of `ZV` as depicted in Fig. 1-b. When $\widehat{v} = 0$ (the counter is not counting), nothing happens as it is accurately presented in Fig. 1-c.

[5] For presentation reasons, we have substituted `ZV` + 1 by `ZV`.

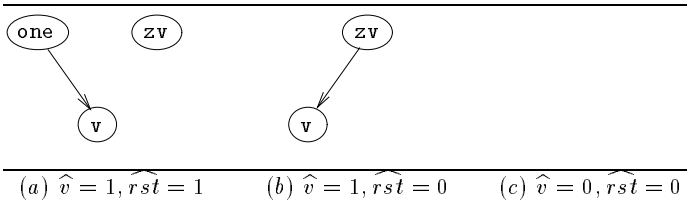


Fig. 1. The Data-Dependencies According to the Control States

The abstract representation of the flows of data using Synchronous-Flow Dependence Graphs (SFD Graphs) is defined by superimposing all the possible data-dependence graphs. Superimposing all the data-dependence graphs drawn in Fig. 1 induces the SFD graph depicted in Fig. 2.

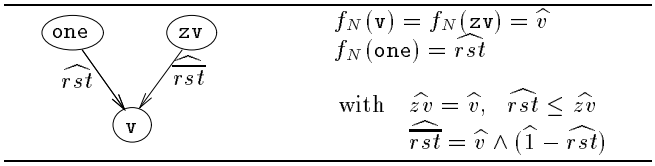


Fig. 2. A Synchronous-Flow Dependence Graph

The paths taken by the data according to the control states are described over SFD graphs by means of two mappings f_N and f_Γ. These two mappings respectively label its nodes and its vertices:

- $f_N(\mathtt{one}) = \widehat{rst}$ means that `ONE` is only present when `RST` occurs[6];
- $f_\Gamma(\mathtt{zv}, \mathtt{v}) = \widehat{\overline{rst}}$ means that `V` is defined from `ZV` when `RST` does not occur.

The new clock-label $\widehat{\overline{rst}}$ denotes the control state (b) in Fig. 1: $\widehat{\overline{rst}} = 1$ when $\widehat{v} = 1$ and $\widehat{rst} = 0$. The definition of $\widehat{\overline{rst}}$ is[7]: $\widehat{\overline{rst}} = \widehat{v} \wedge (\widehat{1} - \widehat{rst})$.
Formally, a SFD graph is defined by:

$< G, C, \Sigma, f_N, f_\Gamma >$ is a Synchronous-Flow Dependence Graph (SFD graph) iff:

- *$G =< N, \Gamma, I, O >$ is a dependence graph $< N, \Gamma >$ with communication nodes: the inputs I and the outputs O are such that $I \subset N, O \subset N$ and $I \cap O = \emptyset$.*
- *$< C, \Sigma >$ is an equational control representation where Σ is a set of constraints over a set C of characteristic functions called clocks;*
- *$f_N : N \longrightarrow C$ is a mapping labeling each node with a clock; it specifies the existence condition of the nodes.*
- *$f_\Gamma : \Gamma \longrightarrow C$ is a mapping labeling each edge with a clock; it specifies the existence condition of the edges.*

[6] Note that the clock of a constant signal is defined in a demand-driven way.
[7] $\widehat{\overline{rst}}$ is not equivalent to $\overline{\widehat{rst}}$ which is the complementary of $\widehat{rst}$: $\overline{\widehat{rst}} = \widehat{1} - \widehat{rst}$

Directed Acyclic Graphs (DAGs) are a very common abstract program representation [1] of the flows of data which may occur in a program. A SFD graph is nothing but a set of directed graphs packed together, the way these graphs are packed being described by a boolean labeling of the elements of this graph. For this reason, we say that SFD graphs are a generalization of DAGs. In contrast with DAGs, the clock labeling provides SFD graphs with a dynamical feature. To express precedence constraints, this clock labeling imposes two constraints which are implicit for DAGs:

- *an edge cannot exist if one of its extremity nodes does not exist.*
  This property translated into the clock algebra, the image set of the mappings, is:

$$\forall(\mathtt{x},\mathtt{y}) \in \Gamma \qquad f_\Gamma(\mathtt{x},\mathtt{y}) \;\leq\; f_N(\mathtt{x}) \wedge f_N(\mathtt{y})$$

- *a cycle of dependencies stands for a deadlock.*
  This property is verified over DAGs by definition. Over SFD graphs, it is expressed as:

  A SFD graph $< G, C, \Sigma, f_N, f_\Gamma >$ *is deadlock free iff,*

  for every cycle $\mathtt{x1}, \ldots, \mathtt{xn}, \mathtt{x1}$ *in* G,

$$f_\Gamma(\mathtt{x1},\mathtt{x2}) \wedge f_\Gamma(\mathtt{x2},\mathtt{x3}) \wedge \ldots \wedge f_\Gamma(\mathtt{xn},\mathtt{x1}) = \widehat{0}$$

  Intuitively, this equation translates the property that a deadlock does not exist if all the dependencies of a cycle in a SFD graph cannot be present at the same time.

As SIGNAL is a dataflow language, SFD graphs define naturally a fine-grain parallel representation of programs. Implementing SIGNAL programs onto a parallel architecture needs to tune the grain of the abstract program representation according to the target parallel architecture. For this purpose, we present in the next section the notion of abstraction over SFD graphs. This notion of abstraction frees SFD graphs from the fine-grain representation they were initially bound.

4 Abstraction of Synchronous-Flow Dependence Graphs

A key concept in software engineering is the concept of abstraction [10] which supplies the sufficient information to compose processes leaving aside any internal feature: it is the key concept for *modularity*. In programming languages, an abstracted process if often confined to an identifier and a set of input/output nodes. In some high-level languages, process abstraction may include (a) formal parameter to introduce some program genericity or (b) some high-order inputs like the procedure entry level in ADA [19].

More generally, the concept of abstraction is designed for the verification of global properties by the composition of synthesized representations. If R_1 and R_2

are two representations with some semantics and | is a composition operator, the *Abs* synthesizing mechanism for the compositional verification of the property P must verify the following relation:

$$P(R_1) \wedge P(R_2) \wedge P(Abs(R_1)|Abs(R_2)) \Longrightarrow P(R_1|R_2)$$

According to their mixed nature, abstraction of SFD graphs involves two synthesizing mechanisms to verify deadlock freedom and to perform control consistency [15] by composition:

- *A synthesis of the internal dependencies*
  This synthesis, required to verify deadlock by composition, is achieved through the transitive closure of the dependence graph and its projection (sub-graph) upon the input and output nodes. The transitive closure of SFD graphs is simply computed with the two following rules.

  rule of series $\quad \mathtt{x} \xrightarrow{\widehat{c}} \mathtt{y} \xrightarrow{\widehat{d}} \mathtt{z} \Rightarrow \mathtt{x} \xrightarrow{\widehat{c} \wedge \widehat{d}} \mathtt{z}$

  rule of parallel $\quad \left.\begin{array}{l} \mathtt{x} \xrightarrow{\widehat{c}} \mathtt{y} \\ \mathtt{x} \xrightarrow{\widehat{d}} \mathtt{y} \end{array}\right\} \Rightarrow \mathtt{x} \xrightarrow{\widehat{c} \vee \widehat{d}} \mathtt{y}$

- *A clock equation projection.*
  This control projection synthesizes the relations (equivalence, inclusion, exclusion) among the clocks which label (a) the edges of the synthesized graph to enable compositional deadlock detection and (b) the input-output nodes to perform control consistency [15] by composition.

The counter example is too small to illustrate the abstraction of SFD graphs. The reader interested in such an example is referred to [15, 14].

In contrast with a lot of common abstract representations of programs, the abstraction over SFD graphs provides them not with black box abstractions but rather with grey box abstractions since it even synthesizes the control. Moreover, as abstractions of SFD graphs are SFD graphs, all the tools previously defined for SFD graphs are reusable modularly: modularity may be introduced in the whole compilation process. With this notion of abstraction, steps (a) and (d) of our scheduling strategy can be achieved without giving up the SFD graph modeling. This modeling homogeneity warrants a greater reliability (every modeling change constitutes a possible source of error) which is a critical requirement for real-time systems. In the next section, we go one step further towards the inference of time-predictable parallel implementations by means of the notion of compile-time scheduling over abstractions of SFD graphs.

5 Compile-Time Scheduling

Compile-time scheduling, that is programming at compile-time the execution of tasks, can be considered at two levels: at the logical level, compile-time scheduling is to set the precedence constraints verified at run-time among the tasks; at

the physical level, compile-time scheduling is to define the exact firing time of the tasks. In this section, we only consider compile-time scheduling at the logical level since we do not want to introduce quantitative data as required for physical compile-time scheduling.

When an application is abstractly represented as a directed acyclic graph $< N, \Gamma >$, scheduling at compile-time a set N of tasks is specified by adding precedence constraints to Γ while making sure that no deadlock is introduced. If we call *reinforcement* the addition of precedence constraints, and *deadlock consistency* the action of "making sure that no deadlock is introduced", a scheduling of $< N, \Gamma >$ is defined as a deadlock-consistent a reinforcement it. Let us transpose this definition from DAGS to SFD graphs:

- *reinforcement*: $< N, \Gamma' >$ is a reinforcement of $< N, \Gamma >$ iff $\Gamma \subseteq \Gamma'$
  Transposing the reinforcement definition to SFD graphs implies:

  $$\mathtt{x} \xrightarrow{\widehat{k}} \mathtt{y} \quad \textit{is a reinforcement of} \quad \mathtt{x} \xrightarrow{\widehat{h}} \mathtt{y} \quad \textit{iff} \quad \widehat{h} \leq \widehat{k}$$

  Note that the absence of dependency between two nodes can be equivalently represented over SFD graphs by a dependence labeled with the null clock $\widehat{0}$. As for DAGs, reinforcement provides a set of SFD graphs based on the same node set with an order relation.
- *deadlock consistency*: $< N, \Gamma' >$ is deadlock-consistent for $< N, \Gamma >$ iff
  $< N, \Gamma >$ deadlock free $\Longrightarrow < N, \Gamma' >$ deadlock free

  Transposing the notion of deadlock consistency to SFD graphs implies:

  $$\mathtt{x} \xrightarrow{\widehat{k}} \mathtt{y} \quad \textit{is deadlock-consistent for} \quad \mathtt{x} \xrightarrow{\widehat{h}} \mathtt{y} \quad \textit{iff}$$

  $$\forall \mathtt{z1}, \ldots, \mathtt{zn} \in N \text{ such that } \mathtt{y} \xrightarrow{\widehat{l0}} \mathtt{z1} \xrightarrow{\widehat{l1}} \mathtt{z2} \quad \ldots \quad \mathtt{zn} \xrightarrow{\widehat{ln}} \mathtt{x} \ : \quad \bigwedge_{0}^{n} \widehat{li} \wedge \widehat{h} = \widehat{0} \quad \Longrightarrow \quad \bigwedge_{0}^{n} \widehat{li} \wedge \widehat{k} = \widehat{0}$$

  Over a transitive closure or an abstraction of a SFD graph, the above condition of deadlock consistency is rewritten in a simpler form:

  $$\mathtt{x} \xrightarrow{\widehat{k}} \mathtt{y} \quad \textit{is deadlock-consistent for} \quad \mathtt{x} \xrightarrow{\widehat{h}} \mathtt{y} \quad \textit{iff}$$

  $$\widehat{h} \wedge \widehat{l} = \widehat{0} \quad \Longrightarrow \quad \widehat{k} \wedge \widehat{l} = \widehat{0} \qquad \text{with} \quad \mathtt{y} \xrightarrow{\widehat{l}} \mathtt{x}$$

  As for DAGs, deadlock consistency provides a set of deadlock-free SFD graphs based on the same node set with an order relation.

A compile-time scheduling of a graph is defined as a *deadlock-consistent reinforcement* of it. Let us focus on what precisely means the combination of these two properties over the SFD graph abstraction depicted in Fig. 3. In this figure, x and y stand for any two nodes, they may be internal, input or output nodes.

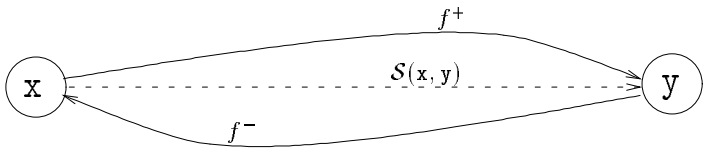


Fig. 3. A Basic SFD Graph Abstraction

The clock which labels the dependency from $\mathtt{x}$ to $\mathtt{y}$ is denoted f^+, its converse is denoted f^-. $\mathcal{S}(\mathtt{x},\mathtt{y})$ stands for a logical $\mathcal{S}$cheduling of $\mathtt{x}$ before $\mathtt{y}$.
As a scheduling is a deadlock-consistent reinforcement, $\mathcal{S}(\mathtt{x},\mathtt{y})$ must ensure that the cycle $\mathtt{x} \xrightarrow{\mathcal{S}(\mathtt{x},\mathtt{y})} \mathtt{y} \xrightarrow{f^-} \mathtt{x}$ does not represent a deadlock. Therefore, it must satisfy the condition (1).

$$\mathcal{S}(\mathtt{x},\mathtt{y}) \wedge f^- = \widehat{0} \tag{1}$$

By combining reinforcement with deadlock consistency, we demonstrate that $\mathcal{S}(\mathtt{x},\mathtt{y})$ defines a compile-time scheduling iff the condition (2) holds.

$$f^+ \leq \mathcal{S}(\mathtt{x},\mathtt{y}) \leq \widehat{x} \wedge \widehat{y} \wedge (\widehat{1} - f^-) \tag{2}$$

Proof: the lowerbound of scheduling is the straightforward expression of the reinforcement property which is attached to the notion of compile-time scheduling. The upperbound of scheduling is induced from the conjunction of the deadlock consistency condition with the inclusion condition. The inclusion condition bound to SFD graph imposes that an arc cannot exist if one of its extremity node does not. Over the notations of Fig. 3, this inclusion condition is translated as: $\mathcal{S}(\mathtt{x},\mathtt{y}) \leq \widehat{x} \wedge \widehat{y}$.
By means of elementary clock calculus, the deadlock consistency property is rewritten as an inequation:

$$\begin{aligned} \mathcal{S}(\mathtt{x},\mathtt{y}) \wedge f^- = \widehat{0} \quad &\Leftrightarrow \quad \mathcal{S}(\mathtt{x},\mathtt{y}) \wedge (\widehat{1} - f^-) = \mathcal{S}(\mathtt{x},\mathtt{y}) \\ &\Leftrightarrow \quad \mathcal{S}(\mathtt{x},\mathtt{y}) \leq (\widehat{1} - f^-) \end{aligned}$$

■

The intuitive meaning of this formally proven upperbound of scheduling is:

$\mathtt{x}$ may be schedule before $\mathtt{y}$ at most when

$\widehat{x} \wedge \widehat{y}$	$\mathtt{x}$ and $\mathtt{y}$ are present,
$\wedge(\widehat{1} - f^-)$	and $\mathtt{y}$ does not precede $\mathtt{x}$

By means of clock expressions, different kinds of scheduling may be expressed at compile-time. If $\mathcal{S}(\mathtt{x},\mathtt{y})$ is equal to $\widehat{x} \wedge \widehat{y}$, it expresses that $\mathtt{x}$ is scheduled before $\mathtt{y}$ as soon as $\mathtt{x}$ and $\mathtt{y}$ are defined: the underlying scheduling is static. The existence of cycle such that $\mathcal{S}(\mathtt{x},\mathtt{y}) \wedge \mathcal{S}(\mathtt{y},\mathtt{x}) = \widehat{0}$ denotes a scheduling depending on boolean conditions evaluated at run-time, it induces pre-constrained dynamic scheduling. The lack of dependency between $\mathtt{x}$ and $\mathtt{y}$, which occurs when $\mathcal{S}(\mathtt{x},\mathtt{y})$ and $\mathcal{S}(\mathtt{y},\mathtt{x})$ are both equal to $\widehat{0}$, induces a dynamic scheduling.

Since scheduling is defined as the conjunction of reinforcement with deadlock consistency, it provides a set of deadlock free SFD graphs based on the same

node set with an order relation. Therefore, an execution schema can be designed progressively by successive reinforcement of a graph. Moreover, this design can be performed at any level of abstraction since this scheduling is applicable over SFD graph abstractions.

Besides the proper definition of the notion of compile-time scheduling over SFD graphs, the purpose of this section was to illustrate the way to express by clock expressions the control of the execution of processes. The same technique is used in the next section to define the notion of compositional deadlock consistency on which our clustering algorithms are based.

6 Compositional Deadlock Consistency

The general problem of partitioning/mapping an application graph onto a set of processors while minimizing the maximal completion time is NP-complete [18]. Bypassing this complexity can be achieved through clustering heuristics which detect properties of sub-graphs that are considered as atomic unit for the mapping process. A clustering phase is intended to increase the granularity of the graph thereby reducing the size of the mapping problem without compromising the implementation efficiency. Then, on this size-reduced application graph, mapping algorithms with higher complexities can be reasonably used.

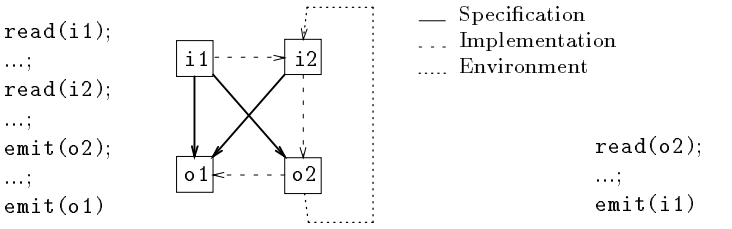
With respect to subtle variations of the scheduling goals, several clustering heuristics have been defined in the literature —see [6] for a survey of these heuristics. The clustering sub-goals that are used can be split in two classes: the quantitative goals (called performance goals in [6]) and the qualitative ones. Two different quantitative data are usually added to the application graph $< N, \Gamma >$ for quantitative scheduling: the execution time e_{ik} of the task n_i ($n_i \in N$) on the processor P_k, and the communication cost c_{ij} between the tasks n_i and n_j when they are mapped on two directly connected processors (null communication time is assumed if n_i and n_j are mapped on the same processor). According to these two kinds of quantitative data, the two extreme sub-goals are: the maximization of the execution efficiency and the minimization of the communication volume. In contrast with the quantitative goals which are architecture dependent, the qualitative goals focus on the shape of the clusters. The qualitative goals which have been used for clustering include:

- *linearity* [9]. A linear cluster is a set of nodes in which, for every couple of nodes, one precedes the other; the nodes of a linear cluster belong to a single path in the dependence graph. As linear clustering merges only sequentially executable nodes, it preserves the parallelism embedded in the graphs;
- *convexity* [18]. Sarkar defines the convexity as the property that ensures that a macro-actor can run to completion once all its input are available. In other words, its execution can be split into three periods sequentially performed: waiting for all the inputs; computing; emitting all the outputs. Therefore, we say that the execution of convex macro-actors is function-like at the I/O level. A graph-theoretic approach to convexity has been studied in [12].

In this section, we define a new qualitative criterion, namely *Compositional Deadlock Consistency*, which allows one to encompass linear as well as convex clustering in a single framework. This extension has been motivated by the reactive feature of real-time systems which imposes to consider the environment of the real-time systems at all their design stages.

6.1 Example

Let us consider the graph in Fig. 4 which depicts the abstraction of a process with two input signals `I1` and `I2`, and two outputs `O1` and `O2`. In this graph, the solid arrows (`i1` → `o1`, `i1` → `o2` and `i2` → `o1`) represent the dependencies induces from the abstraction of the specification of the process.



(a) Implementation (b) Composition graph (c) Environment

Fig. 4. A Deadlock between a Process Implementation and its Environment

A topological sort of these nodes may induce the static scheduling in Fig. 4-a. Transposing this static scheduling over the graph in Fig. 4-b introduces the dashed arrows. If we compose the implementation in Fig. 4-a with an environment implementing the scheme in Fig. 4-c, a deadlock is created. At the graph level, this deadlock is denoted by the cycle `i2` - - -> `o2` ·····> `i2` . This deadlock is present at the implementation level but not at the specification level since it includes a dashed arrow. As the scheduling `i2` - - -> `o2` may create a deadlock with an environment which is correct with respect to the process specification, this scheduling is said *not compositionally deadlock-consistent.*

In contrast, the scheduling `o2` - - -> `o1` is compositionally deadlock-consistent since it does not create a deadlock with the environment in Fig.4-c, and this environment is the only one which can read outputs and emit inputs of the process without creating a deadlock with it at the specification level.

6.2 Definition

Let us focus on what precisely means the notion of *compositional deadlock consistency* over the generic SFD graph abstraction depicted in Fig. 5.

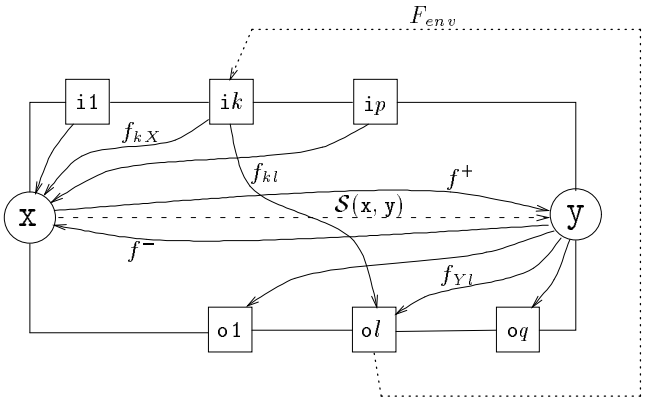


Fig. 5. A Generic SFD Graph Abstraction

In this figure, $\mathtt{i}1 \cdots \mathtt{i}p$ represent the input nodes, $\mathtt{o}1 \cdots \mathtt{o}q$ the output ones and, $\mathtt{x}$ and $\mathtt{y}$ stand for any two nodes which may be internal nodes as well interface nodes[8]. Translated over the notations in Fig. 5, the notion of compositional deadlock consistency imposes that $\mathcal{S}(\mathtt{x},\mathtt{y})$ must verify:

$$\forall \mathtt{i}k, \mathtt{o}l \qquad f_{kX} \wedge \mathcal{S}(\mathtt{x},\mathtt{y}) \wedge f_{Yl} \wedge F_{env} = \widehat{0} \tag{3}$$

In this equivalence, F_{env} denotes a dependency from $\mathtt{o}l$ to $\mathtt{i}k$ outcoming from the composition with an environment. This environment is acceptable if it is not deadlocked with the specification of the process. Thus, the following condition must be verified:

$$\forall \mathtt{i}k, \mathtt{o}l \qquad F_{env} \wedge f_{kl} = \widehat{0}$$

As for the proof of the upperbound of scheduling (formula (2) in section 5), the above condition can be equivalently rewritten in the inequation $F_{env} \leq (\widehat{1} - f_{kl})$. Consequently, the condition of compositional deadlock consistency (formula (3)) is rewritten in:

$$\forall \mathtt{i}k, \mathtt{o}l \qquad \mathcal{S}(\mathtt{x},\mathtt{y}) \wedge f_{kX} \wedge f_{Yl} \wedge (\widehat{1} - f_{kl}) = \widehat{0}$$

This quantified equation can be rewritten in inequation (4).

$$\mathcal{S}(\mathtt{x},\mathtt{y}) \leq \bigwedge_{k,l} (\widehat{1} - f_{kX} \wedge f_{Yl} \wedge (\widehat{1} - f_{kl})) \tag{4}$$

Proof: The equation $\mathcal{S}(\mathtt{x},\mathtt{y}) \wedge f_{kX} \wedge f_{Yl} \wedge (\widehat{1} - f_{kl}) = \widehat{0}$ can be rewritten in:

$$\begin{aligned}
&\mathcal{S}(\mathtt{x},\mathtt{y}) \wedge (\widehat{1} - f_{kX} \wedge f_{Yl} \wedge (\widehat{1} - f_{kl})) = \mathcal{S}(\mathtt{x},\mathtt{y}) && \forall \mathtt{i}k \in I, \mathtt{o}l \in O \\
\Leftrightarrow\ &\mathcal{S}(\mathtt{x},\mathtt{y}) \leq (\widehat{1} - f_{kX} \wedge f_{Yl} \wedge (\widehat{1} - f_{kl})) && \forall \mathtt{i}k \in I, \mathtt{o}l \in O \\
\Leftrightarrow\ &\mathcal{S}(\mathtt{x},\mathtt{y}) \leq \textstyle\bigwedge_{k,l} (\widehat{1} - f_{kX} \wedge f_{Yl} \wedge (\widehat{1} - f_{kl}))
\end{aligned}$$

■

[8] If $\mathtt{x}$ is the input node $\mathtt{i}k$, it is equivalent to consider for the sequel of this paper that f_{kX} is equal to $\widehat{x}$. A symmetric remark can be expressed if $\mathtt{y}$ is the output node $\mathtt{o}l$.

6.3 Fully Deadlock Consistent Compile-Time Scheduling

By combining the compile-time scheduling characterization (formula (2)) with inequality (4), we define the criterion of *fully deadlock consistent compile-time scheduling (fdc scheduling)* which is formally characterized by:

$\mathcal{S}(\mathtt{x},\mathtt{y})$ *is defines a fully deadlock consistent compile-time scheduling of* $\mathtt{x}$ *before* $\mathtt{y}$ *iff* $\quad f^+ \leq \mathcal{S}(\mathtt{x},\mathtt{y}) \leq \mathcal{S}^\top(\mathtt{x},\mathtt{y})$ *with:*

$$\mathcal{S}^\top(\mathtt{x},\mathtt{y}) = \widehat{x} \wedge \widehat{y} \wedge (\widehat{1} - f^-) \wedge \bigwedge_{k,l} (\widehat{1} - f_{kX} \wedge f_{Yl} \wedge (\widehat{1} - f_{kl})) \qquad (5)$$

The proof of this inequality is straighforward. The complex clock expression which specifies the upperbound of scheduling may be intuitively read as:

$\mathtt{x}$ may be scheduled before $\mathtt{y}$ iff

$\mathtt{x}$ does not precede $\mathtt{y}$ and	:	$\widehat{x} \wedge \widehat{y} \wedge (\widehat{1} - f^-) \wedge$
if a scheduling path $\mathtt{i}k, \mathtt{x}, \mathtt{y}, \mathtt{o}l$ is created	:	$\bigwedge_{k,l} (\widehat{1} - f_{kX} \wedge f_{Yl} \wedge$
then $\mathtt{i}k$ precedes $\mathtt{o}l$ by specification		$(\widehat{1} - f_{kl}))$

The two main promising properties of this scheduling criterion are: (a) it may induce *architecture independent clustering* since it is a qualitative scheduling criterion; (b) as it is based on the abstraction of SFD graphs, it may be applied to any subset of nodes: it defines an *any level scheduling criterion*. Exploiting these properties to perform clustering needs to use this criterion accurately to avoid the NP-complete problems that its general use will encounter. The practical uses of this new scheduling criterion for clustering are presented in the next section.

7 Clustering

By applying the fdc scheduling criterion to a set of nodes, the associated process may constitute a cluster by:

- *Linear Clustering* if all the nodes may belong to a single path of fdc scheduling. Note that, as a fdc scheduling is a reinforcement of a graph, any linear clustering over a graph (as performed in [9]) is a linear clustering over a fdc scheduling of this graph. But, in contrast with Kim & Browne's linear clustering, linear clustering over fdc scheduled graphs may reduce the parallelism embedded in the initial graph;
- *Convex Clustering* if all the nodes may belong to a single path of fdc scheduling where inputs and outputs are not alternating. Therefore, any convex cluster is a linear cluster.

The practical use of the fdc scheduling criterion to do linear and convex clustering will encounter NP-problems at two levels:

- complex calculi in a boolean algebra lead to NP-complete problems. This first obstacle has been overcome with the heuristic algorithm that implements the clock calculus [2]. Despite the breakthrough achieved by this heuristic algorithm, the boolean calculi submitted to it must be as simple as possible.
- optimal partitioning/clustering of general graphs with respect to non trivial criteria is a NP-complete problem. To cope with this obstacle, we can use optimal algorithms with exponential complexity on very small (sub-)graphs, polynomial but often sub-optimal algorithms on large graphs, or a combination of both.

The first optimization achieved by both the convex and the linear clustering algorithms is to do clustering in two steps. Firstly, only a size-reducted problem is considered by restricting the scope of fdc scheduling from any pair of nodes to pairs of interface nodes. Secondly, the properties detected at the interface level are propagated to the internal nodes to perform convex and linear clustering.

7.1 Convex Clustering

A naive algorithm for convex clustering at the interface level would be to enumerate the possible elementary paths of the maximal fdc schedulings of an interface abstraction, the maximal fdc schedulings of a graph being computed by recursively substituting each arc by its upperbound of fdc scheduling. If one of these paths does not alternate inputs and outputs, the associated process may define a convex cluster.

The major drawback of this naive algorithm is its complexity: it requires two phases (computation of the maximal fdc scheduling and path enumeration) which have an exponential complexity in the general case. Consequently, we have investigated the other possibility which goes through the upperbound of fdc scheduling of an interface graph. The upperbound of fdc of a graph is computed by substituting in parallel each arc by its upperbound of fdc scheduling. This upperbound is the superimposition of all the maximal fdc schedulings. For instance, let us consider the interface abstraction depicted in Fig. 6-a. In this abstraction, we assume that $f_N(\texttt{i1}) = f_N(\texttt{i2}) = f_N(\texttt{o1}) = f_N(\texttt{o2}) = \widehat{k}$ and $\widehat{h} \leq \widehat{k}$. The upperbound of fdc scheduling of this abstraction is the SFD graph in Fig. 6-b.

In the general case, the upperbound of fdc scheduling does not define a scheduling as it may include cycles representing deadlocks. The upperbound of scheduling depicted in Fig. 6-b includes two of these cycles, one between `i1` and `i2` and the other between `o1` and `o2`. The conjunction $\widehat{h} \wedge \widehat{k}$ of the clocks labeling the dependencies of these cycles is equal to $\widehat{h}$ since $\widehat{h} \leq \widehat{k}$: the cycles exist at $\widehat{h}$. In contrast with these two first cycles, the third elementary cycle which occurs between `i2` and `o2` does not stand for a deadlock since $\widehat{h} \wedge (\widehat{k} - \widehat{h}) = \widehat{0}$. This remark is in fact a general property as proved in [14]:

no deadlock cycle including inputs and outputs may occur at the upperbound of fdc scheduling.

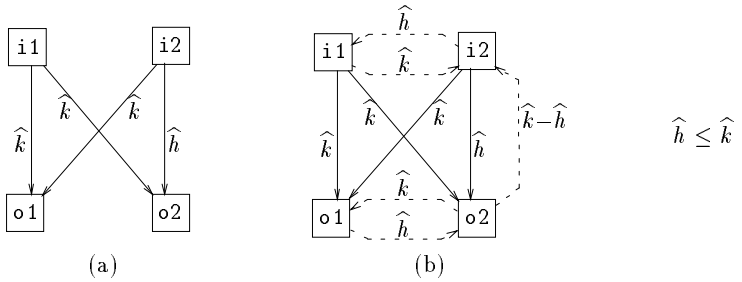


Fig. 6. A Graph and its Upperbound of Fdc Scheduling

As no cycle may alternate inputs and outputs, a cycle among inputs induces that these inputs can be scheduled in a sequence without outputs; a similar discussion may occur for cycles among outputs. This property of the cycles occurring at the upperbound of fdc scheduling motivates the following algorithm which performs convex clustering:

1. compute the upperbound of fdc scheduling among the inputs;
2. for each set of inputs belonging to a cycle at $\widehat{h}$: cluster to this set of inputs the internal and outputs nodes which depend exclusively on these inputs.

Note that a symmetric convex clustering algorithm may start from the outputs instead of the inputs. This variation of the clustering algorithm may be useful if there is less outputs than inputs to deal with a smaller problem. Applied to the interface graph in Fig. 6-a, this algorithm detects that the associated process defines a convex cluster at $\widehat{h}$.

7.2 Linear Clustering

By convex clustering may result a partition into processes which can run to completion once all their inputs are available; in these processes, all the inputs may precede all the outputs at the implementation level. Looking for a partition into linear clusters which are not convex clusters leads to search for fdc scheduling paths which alternate inputs and outputs. Therefore, one way to reduce the search space of the algorithm which does this search is to start from a fdc scheduling dependency connecting an output to an input. Starting from such a scheduling dependency, the algorithm may proceed by looking backward and then forward to get the longest path of fdc scheduling. Previously to this algorithm, a transitive reduction algorithm may be applied to reduce even more the search space.

Applied to the graph in Fig. 6-a, the algorithm starts from the fdc scheduling dependency `o2---> i2` at $\widehat{k} - \widehat{h}$. Then, by going backward, the node `i1` is added as the starting point of this scheduling path. By going forward, the node `o1` is appended to the path. Finally, this algorithm detects that the associated process defines a linear cluster but not a convex one at $\widehat{k} - \widehat{h}$. By combining this

result with the convex clustering detected on the same set of nodes, linear and possibly convex cluster are detected. By this combination, the process abstractly represented in Fig. 6-a defines a linear cluster at:

$$\widehat{h} \vee (\widehat{k} - \widehat{h}) = \widehat{k}$$

After this definition of the convex and linear clustering algorithms, let us conclude this paper by presenting the way we intend to implement the five-steps scheduling strategy we advocated, and how the fdc scheduling criterion is used in this framework.

7.3 Scheduling Strategy Implementation

In the beginning of this paper, we advocate a five-steps scheduling strategy.

(a). Gather the nodes of the application graph into u clusters ($u \geq p$).
(b). Merge the u clusters into p connected virtual processors.
(c). Map the p virtual processors onto the p physical processors.
(d). Partition each virtual processor i in v_i clusters.
(e). Compute a static schedule for each cluster; the resulting sequences of code will be dynamically scheduled.

The two clustering steps (a) and (d) will be based on the convex and linear clustering algorithms previously presented. Steps (b) and (c) will be implemented by means of the coupling of the SIGNAL software design environment with the SYNDEX system. SYNDEX, which stands for SYNchronous Distributed EXecutive, is a system which enables the inference of implementations over various distributed architectures. It performs this inference by mapping SFD graphs over a graph representation of the architecture[9]. This inference is performed in three steps: (a) the user may constrain some mapping of processes onto processors; (b) SYNDEX completes the mapping and produces scheduled distributed code for the target architecture and (c) SYNDEX provides the user with static analyses of the performance of the inferred implementation. An iteration among these three steps is required to infer for complex applications an efficient implementation on a distributed, eventually heterogeneous, architecture.

Implementing step (e) may take once again benefit of the fdc scheduling criterion but in a slighly different way than it has been achieved for clustering. Defining an implementation requires an order relation. From the upperbound of fdc scheduling of an interface graph, two ways exist to get an order relation: break the cycles or merge the nodes belonging to a cycle. Using these two methods over the upperbound in Fig. 6-b, we infer the two graphs in Fig. 7 which respectively define:

[9] In fact, the graph representation of the architecture may be an hypergraph since the target architecture may include buses.

– an *interface execution scheme* in Fig. 7-a.
The first step in the inference of this high-level execution scheme in Fig. 7-a is to break the cycles representing deadlocks by removing the edges between inputs and between outputs at the clock at which convex clustering was performed. This leads to suppress in Fig 6-b the arcs `i2`---> `i1` and `o1`---> `o2`. The second step is the unfolding of the acyclic graph according to the different control states referred in the remaining cycles. The remaining cycle between `i2` and `o2`, which does not denote a deadlock ($\widehat{h} \wedge (\widehat{k} - \widehat{h}) = \widehat{0}$), imposes a conditional scheduling denoted by the labeled fork-join in the graph in Fig. 7-a. Note that static (i.e. non conditional) scheduling is achieved at the two extreme cases: $\widehat{h} = \widehat{k}$ and $\widehat{h} = \widehat{0}$.
– a *communication scheme* in Fig.7-b.
The cycle between the input nodes expresses that, when $\widehat{h}$ occurs, `i1` may be scheduled before or after `i2` without creating a deadlock. For this reason, the values on `i1` and `i2` can be received gathered without the creation of a deadlock. In other words, the communications of the values of `i1` and `i2` may be vectorized at $\widehat{h}$ if they come from the same processor. To express the design of such a communication scheme at the graph level, it is sufficient to partition the nodes according to the cycles. Applied to the upperbound graph in Fig. 6-b, we may deduce the input communication interface presented in Fig. 7-b. In this implementation, the values carried by `i1` and `i2` are communicated gathered at $\widehat{h}$ through the new node `ci12`: $f_N(\texttt{ci12}) = \widehat{h}$. A symmetric result may be achieved over the outputs.

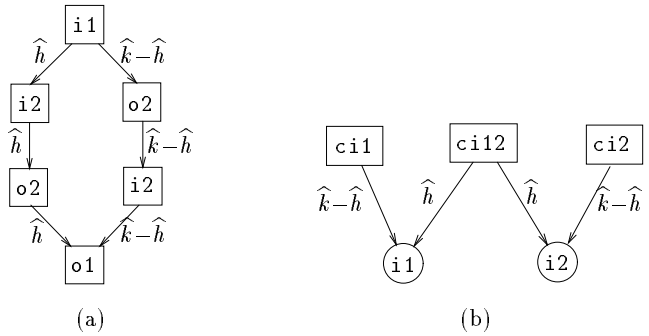


Fig. 7. Execution and Communication Schemes

8 Conclusion

The paper has motivated a scheduling strategy for the distributed implementation of SIGNAL programs. This scheduling strategy differs from the usual one by

the dynamical scheduling it includes. This variation has been motivated by the reactive requirements that SIGNAL, as a real-time language, must fulfill.

For the implementation of this scheduling strategy, we have defined several tools, all of them acting on Synchronous-Flow Dependence Graphs (SFD Graphs). These graphs, which constitute the abstract representation of SIGNAL programs, define a generalization of the notion of Directed Acyclic Graph. Three tools are defined in this paper to implement this scheduling strategy:

- *Abstraction.*
  This first tool is intended to free SFD graphs from the fine-grain parallel abstract representation they were initially bound. By means of this abstraction, we are able to tune the grain-size of the representation according to the one of the target architecture without giving up with the SFD graph modeling;
- *Compile-time Scheduling.*
  This definition of the notion of scheduling constitutes the first step towards the inference of implementations. The purpose of this definition was also to illustrate the method to express over SFD graphs the scheduling of processes with a complex control;
- *Clustering.*
  A new qualitative criterion, namely compositional deadlock consistency, is defined and used at several steps in the scheduling strategy. In particular, this new criterion is used to implement the two clustering steps of our scheduling strategy. This new criterion enable to embraces in a single framework two usual qualitative clustering criteria, linearity and convexity.

The abstraction tool is currently integrated into the SIGNAL software design environment; the programming of the clustering tools is underway. The SIGNAL software design environment intends to encompass all the stages of the design of real-time systems. This environment includes (a) a graphic specification interface to specify real-time systems, (b) several formal verification tools to prove properties thereby to enhance the safety of the implementations and (c) tools to infer implementations over sequential architectures as well as distributed ones.

The inference of distributed implementations for SIGNAL programs is only partially implemented in the SIGNAL compiler; the architecture-dependent transformations are performed by the SYNDEX system [11]. The coupling between the SIGNAL compiler and the SYNDEX system is achieved by means of a textual decompilation of SFD graphs [4]; an extended version of this decompilation defines the common graph format shared by ESTEREL [5], ARGOS [17], LUSTRE [8] and SIGNAL.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wiley, 1986.

2. T. Amagbegnon, L. Besnard, and P. L. Guernic. Aborescent canonical form of boolean expressions. Research Report 826, IRISA, June 1994.
3. S. H. Bokhari. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. on Computers*, 37(1):48–57, January 1988.
4. P. Bournai, C. Lavarenne, P. Le Guernic, O. Maffeïs, and Y. Sorel. Interface SIGNAL-SynDEx. Research report 2206, INRIA France, Rennes, march 1994.
5. F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, Sept. 1991.
6. A. Gerasoulis and T. Yang. A comparison of clustering heuristics for clustering dags on multiprocessors. *Journal of Parallel and Distributed Computing, Special Issues on Scheduling and Load Balancing*, 16(4):276–291, Dec. 1992.
7. A. Gerasoulis and T. Yang. A static-dataflow scheduling tool for scalable parallel architectures. In *Summer School on Scheduling Theory and its applications*, pages 382–417. Chateau de Bonas(Gers), INRIA, Sept. 1992.
8. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1321, Sept. 1991.
9. S. J. Kim and J. C. Browne. A general approach to mapping of parallel computation upon multiprocessor architectures. In *Int. Conf. on Parallel Processing*, volume III, pages 1–8, 1988.
10. C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.
11. C. Lavarenne, O. Segrouchni, Y. Sorel, and M. Sorine. The SYNDEX software environment for real-time distributed systems design and implementation. In *European Control Conference*, volume 2, pages 1684–1689, June 1991.
12. B. Le Goff, P. Le Guernic, and J. Aráoz Durand. Semi-granules and schielding for off-line scheduling. Research Report 1228, INRIA France, Rocquencourt, May 1990.
13. P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, Sept. 1991.
14. O. Maffeïs. *Ordonnancements de graphes de flots synchrones; Application à* SIGNAL. PhD thesis, Université de Rennes 1, France, Jan. 1993.
15. O. Maffeïs and P. Le Guernic. Combining dependability with architectural adaptability by means of the SIGNAL language. In *3rd Int. Workshop on Static Analysis*, pages 99–110. LNCS no 724, Springer-Verlag, Sept. 1993.
16. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.
17. F. Maraninchi. The ARGOS language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, Oct. 1991.
18. V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts, and Pitman Publishing, London, U.K., 1989.
19. USDD. *Reference Manual for the* ADA *Programming Language*. United States, Department of Defense, 1983. ANSI:MIL-STD-1815A-1983.
20. T. Yang and A. Gerasoulis. Pyrros: Static task scheduling and code generation for message-passing multiprocessors. In *Proc. of the 6th ACM Int. Conf. on Supercomputing*, pages 428–437, 1992.

This article was processed using the LaTeX macro package with LLNCS style